Computer Systems

CSE 410 Autumn 2013

22 Disks and File Systems

Slides adapted from CSE 451 material by Gribble, Lazowska, Levy, and Zahorjan

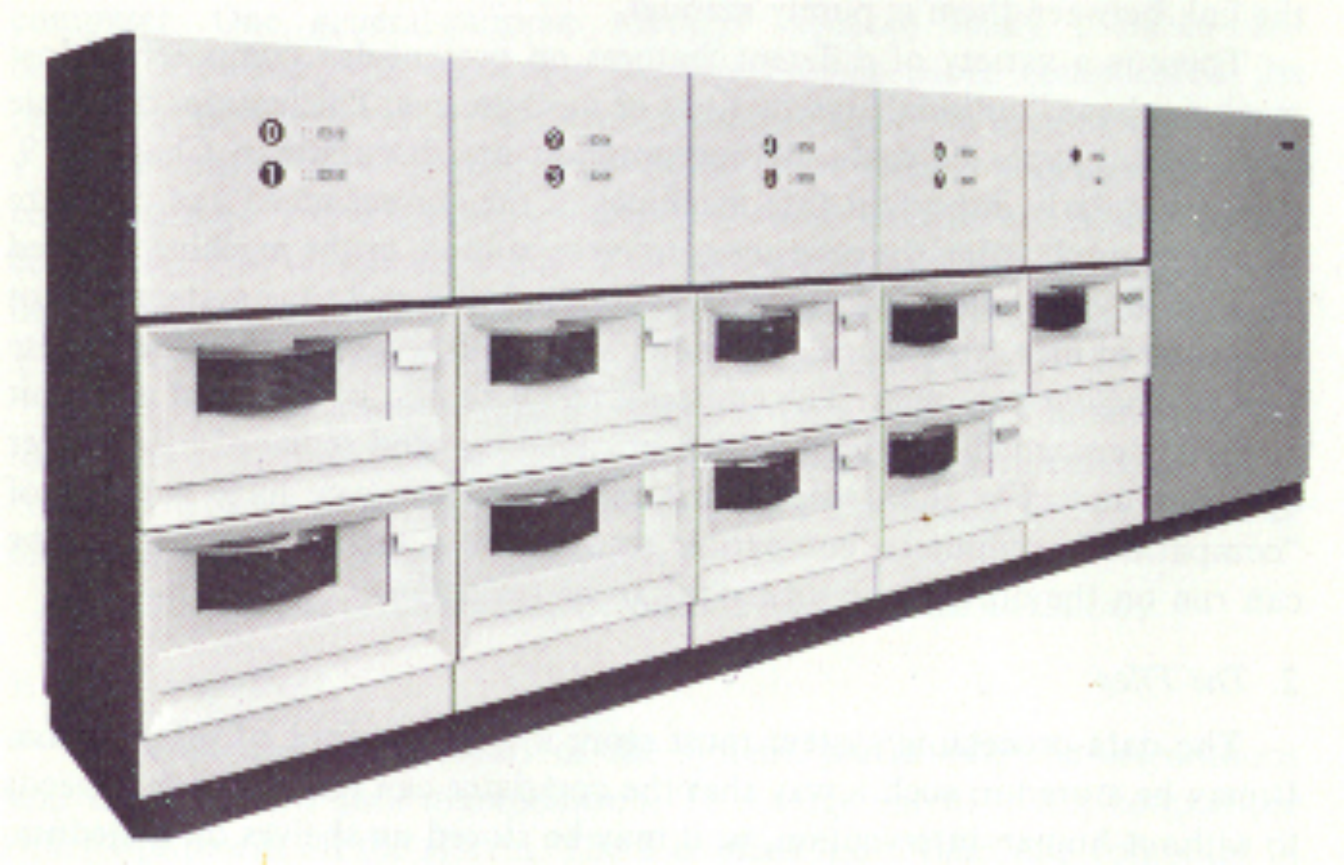
Topics

- **Secondary Storage – Disks**
  - Characteristics
  - Place in memory hierarchy
- **File Systems**
  - Files, directories, and disk blocks
  - Classic Unix file system

Secondary storage

- **Secondary storage typically:**
  - is anything that is outside of “primary memory”
  - does not permit direct execution of instructions or data retrieval via machine load/store instructions
- **Characteristics:**
  - it’s large: 500-2000GB
  - it’s cheap: $0.05/GB for hard drives
  - it’s persistent: data survives power loss
  - it’s slow: milliseconds to access
    - why is this slow??
  - it *does* fail, if rarely
    - big failures (drive dies; MTBF ~3 years)
      - if you have 100K drives and MTBF is 3 years, that’s 1 “big failure” every 15 minutes!
    - little failures (read/write errors, one byte in 10^{13})

A trip down memory lane …



IBM 2314

About the size of

6 refrigerators

8 x 29MB (M!)

Required similar-sized air cond.!

.01% (not 1% – .01%!) the capacity of this $100 4"x6"x1" item

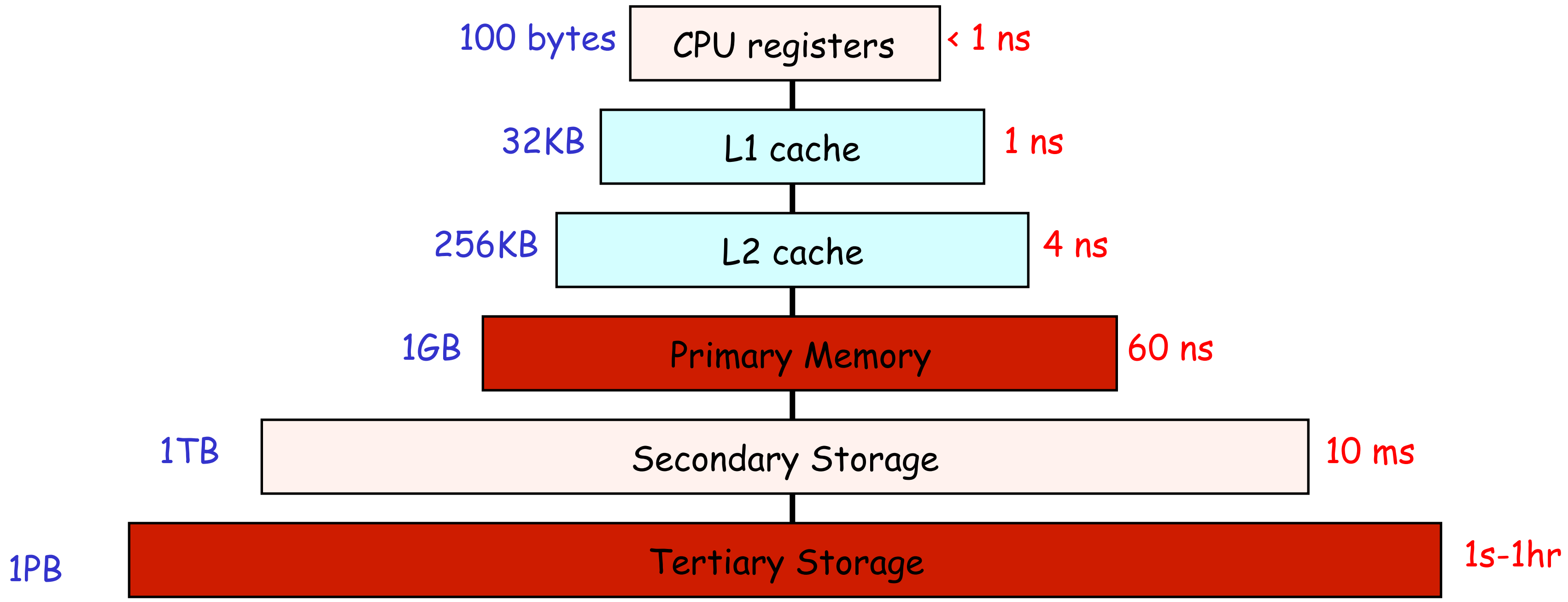
Disk trends

- **Disk capacity, 1975-1989**
  - doubled every 3+ years
  - 25% improvement each year
  - factor of 10 every decade
  - Still exponential, but far less rapid than processor performance
- **Disk capacity, 1990-recently**
  - doubling every 12 months
  - 100% improvement each year
  - factor of 1000 every decade
  - Capacity growth 10x as fast as processor performance!

Disk Trends

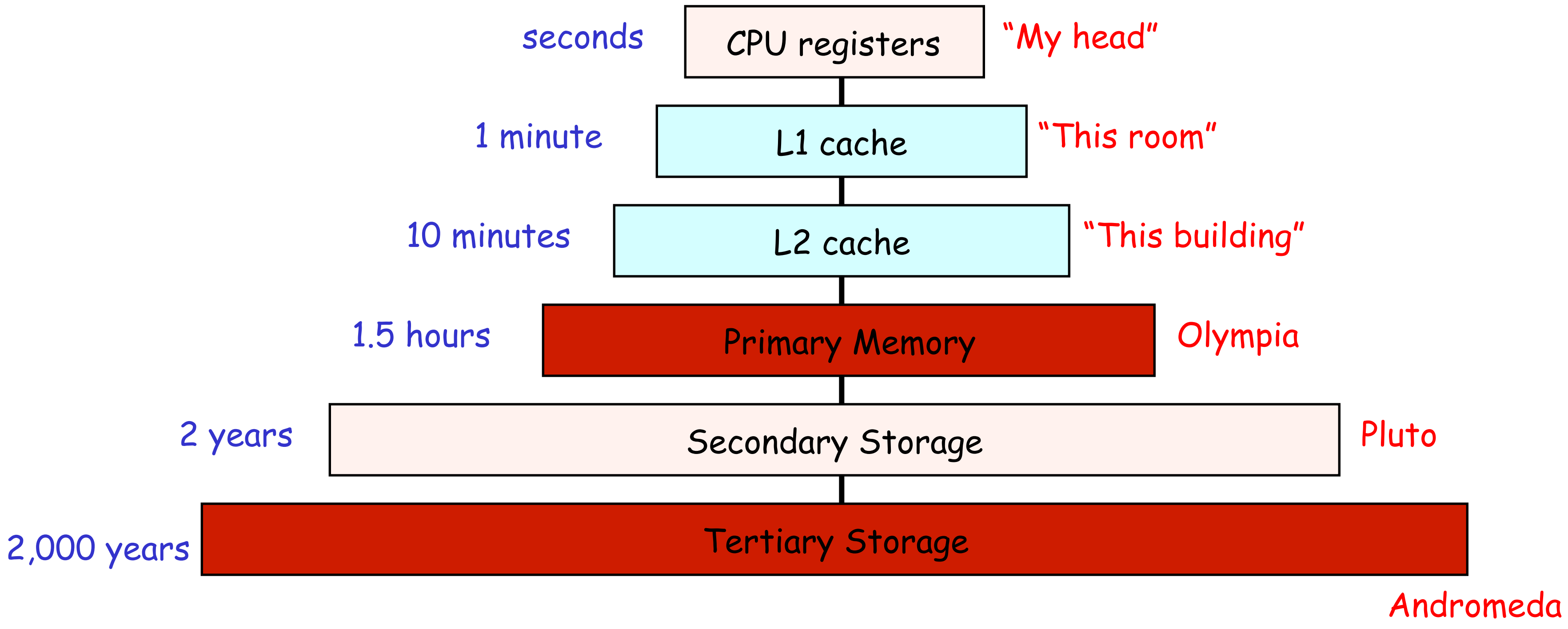
- **Only a few years ago, we purchased disks by the megabyte (and it hurt!)**
- **Today, 1 GB (a billion bytes) costs ~~$1~~ ~~$0.50~~ $0.05 from Dell (except you have to buy in increments of ~~40~~ ~~80~~ ~~250~~ 1000 GB)**
  - => 1 TB costs ~~$1K~~ ~~$500~~ $100, 1 PB costs ~~$1M~~ ~~$500K~~ $100K
- **Technology is amazing**
  - Flying a 747 6" above the ground
  - Reading/writing a strip of postage stamps
- **But …**
  - Jets do crash …

Memory hierarchy

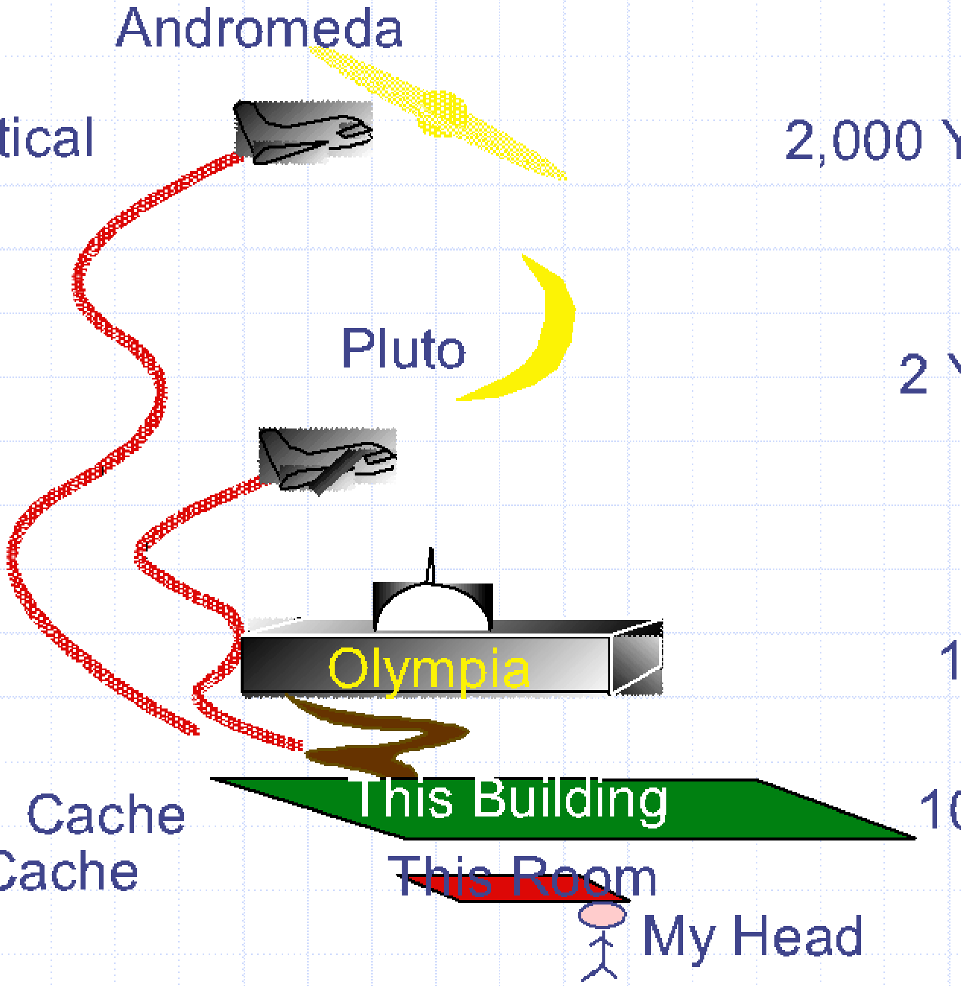


- **Each level acts as a cache of lower levels**

Memory hierarchy: distance analogy



Storage Latency:
How Far Away is the Data?



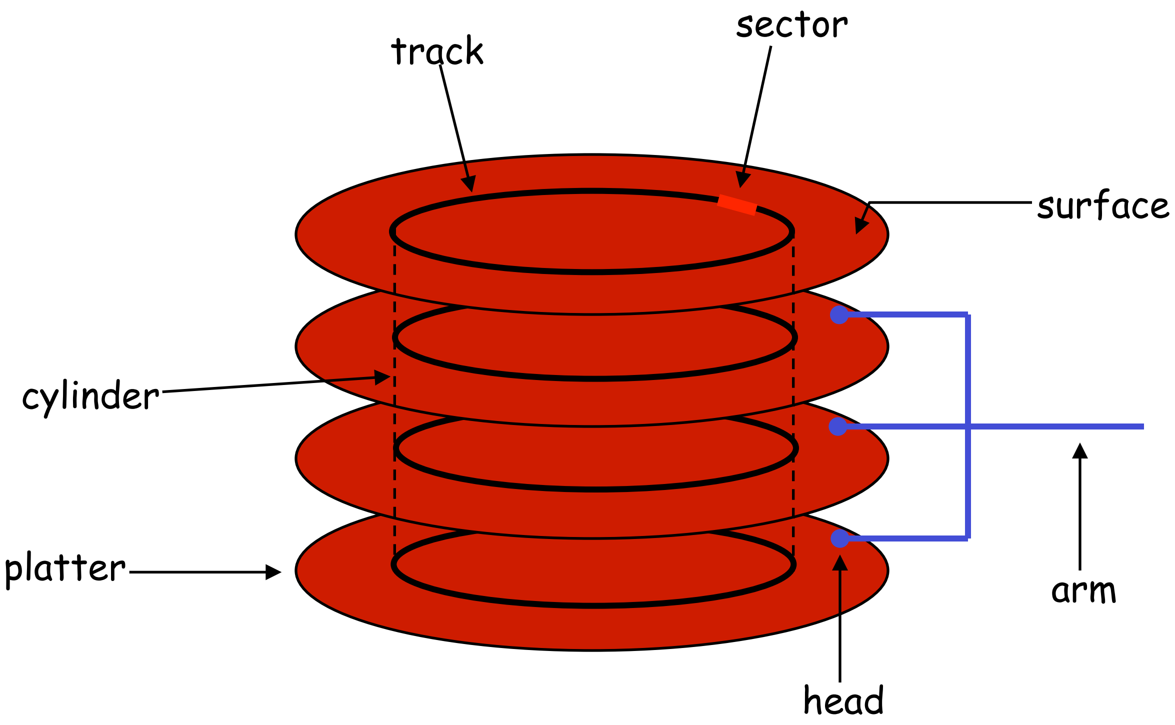
10^9	Tape /Optical Robot	Andromeda	2,000 Years
10^6	Disk	Pluto	2 Years
100	Memory	Olympia	1.5 hr
10	On Board Cache	This Building	10 min
2	On Chip Cache	This Room	
1	Registers	My Head	1 min

Disks and the OS

- **Disks are messy, messy devices**
  - errors, bad blocks, missed seeks, etc.
- **Job of OS is to hide this mess from higher-level software (disk hardware increasingly helps with this)**
  - low-level device drivers (initiate a disk read, etc.)
  - higher-level abstractions (files, databases, etc.)
  - (modern disk drives do some of this masking for the OS)
- **OS may provide different levels of disk access to different clients**
  - physical disk block (surface, cylinder, sector)
  - disk logical block (disk block #)
  - file logical (filename, block or record or byte #)

Physical disk structure

- **Disk components**
  - platters
  - surfaces
  - tracks
  - sectors
  - cylinders
  - arm
  - heads



Disk performance

- **Performance depends on a number of steps**
  - seek: moving the disk arm to the correct cylinder
    - depends on how fast disk arm can move
      - seek times aren't diminishing very quickly (why?)
  - rotation (latency): waiting for the sector to rotate under head
    - depends on rotation rate of disk
      - rates are increasing, but slowly (why?)
  - transfer: transferring data from surface into disk controller, and from there sending it back to host
    - depends on density of bytes on disk
      - increasing, relatively quickly
- **When the OS uses the disk, it tries to minimize the cost of all of these steps**
  - particularly seeks and rotation

Performance via disk layout

- **OS may increase file block size in order to reduce seeking**
- **OS may seek to co-locate "related" items in order to reduce seeking**
    - blocks of the same file
    - data and metadata for a file

Performance via caching, pre-fetching

- **Keep data or metadata in memory to reduce physical disk access**
  - problem?
- **If file access is sequential, fetch blocks into memory before requested**

Performance via disk scheduling

- **Seeks are very expensive, so the OS attempts to schedule disk requests that are queued waiting for the disk**
  - FCFS (do nothing)
    - reasonable when load is low
    - long waiting time for long request queues
  - SSTF (shortest seek time first)
    - minimize arm movement (seek time), maximize request rate
    - unfairly favors middle blocks
  - SCAN (elevator algorithm)
    - service requests in one direction until done, then reverse
    - skews wait times non-uniformly (why?)
  - C-SCAN
    - like scan, but only go in one direction (typewriter)
    - uniform wait times

Interacting with disks

- **In the old days…**
  - OS would have to specify cylinder #, sector #, surface #, transfer size
    - i.e., OS needs to know all of the disk parameters
- **Modern disks are even more complicated**
  - not all sectors are the same size, sectors are remapped, …
  - disk provides a higher-level interface, e.g., SCSI
    - exports data as a logical array of blocks [0 … N]
    - maps logical blocks to cylinder/surface/sector
    - OS only needs to name logical block #, disk maps this to cylinder/surface/sector
    - on-board cache
    - as a result, physical parameters are hidden from OS
      - both good and bad

Seagate Barracuda 3.5” disk drive

- 1Terabyte of storage (1000 GB)
- $100
- 4 platters, 8 disk heads
- 63 sectors (512 bytes) per track
- 16,383 cylinders (tracks)
- 164 Gbits / inch-squared (!)
- 7200 RPM
- 300 MB/second transfer
- 9 ms avg. seek, 4.5 ms avg. rotational latency
- 1 ms track-to-track seek
- 32 MB cache

Solid state drives: imminent disruption

- **Hard drives are based on spinning magnetic platters**
  - *mechanics* of drives determine performance characteristics
    - sector addressable, not byte addressable
    - capacity improving exponentially
    - sequential bandwidth improving reasonably
    - random access latency improving very slowly
  - cost dictated by massive economies of scale, and many decades of commercial development and optimization

Solid State Drives

- **Solid state drives are based on NAND flash memory**
  - no moving parts; performance characteristics driven by electronics and physics – more like RAM than spinning disk
  - relative technological newcomer, so costs are still quite high in comparison to hard drives, but dropping fast

SSD performance: reads

- **Reads**
  - unit of read is a *page*, typically 4KB large
  - today's SSD can typically handle 10,000 – 100,000 reads/s
    - 0.01 – 0.1 ms read latency (50-1000x better than disk seeks)
    - 40-400 MB/s read throughput (1-3x better than disk seq. thpt)

SSD performance: writes

- **Writes**
  - flash media must be *erased* before it can be written to
  - unit of erase is a block, typically 64-256 pages long
    - usually takes 1-2ms to erase a block
    - blocks can only be erased a certain number of times before they become unusable – typically 10,000 – 1,000,000 times
  - unit of write is a page
    - writing a page can be 2-10x slower than reading a page
- **Writing to an SSD is complicated**
  - random write to existing block: read block, erase block, write back modified block
    - leads to hard-drive like performance (300 random writes / s)
  - sequential writes to erased blocks: fast!
    - SSD-read like performance (100-200 MB/s)

SSDs: dealing with erases, writes

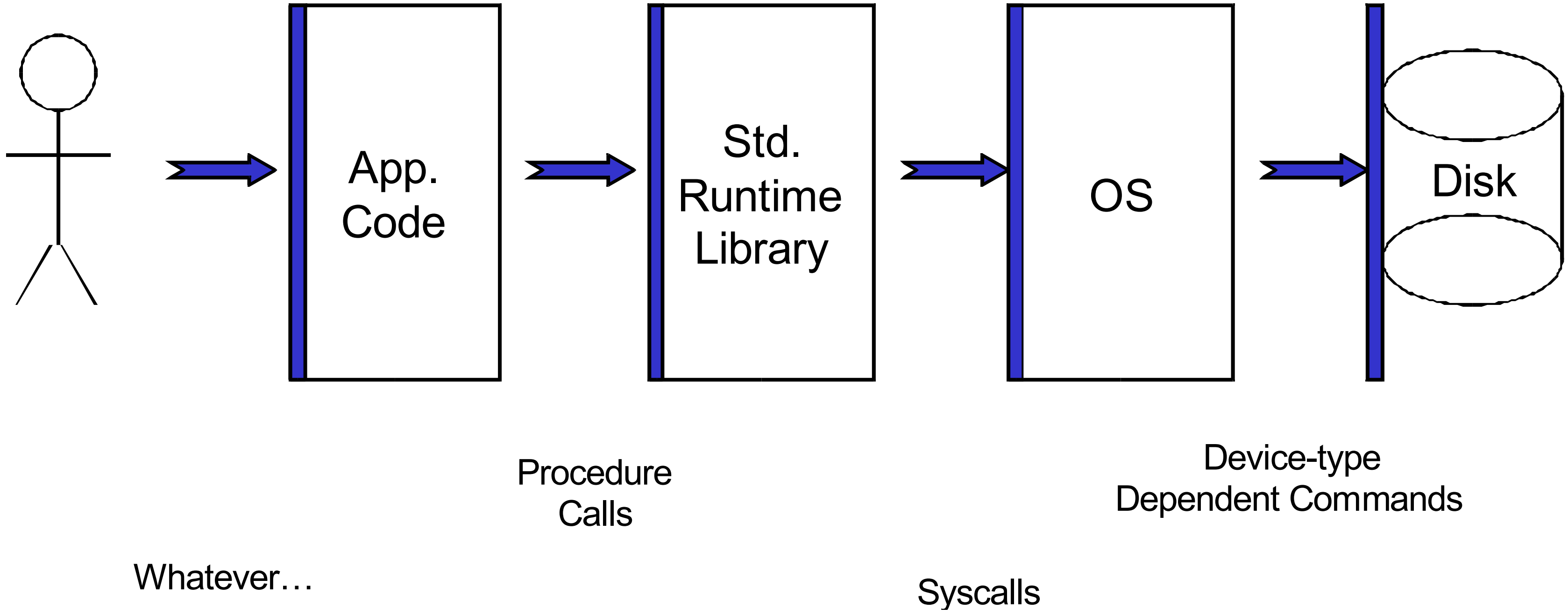
- **Lots of higher-level strategies can help hide the warts of an SSD**
  - many of these work by virtualizing pages and blocks on the drive (i.e., exposing logical pages, not physical pages, to the rest of the computer)
  - wear-leveling: when writing, try to spread erases out evenly across physical blocks of of the SSD
    - Intel promises 100GB/day x 5 years for its SSD drives
  - log-structured filesystems: convert random writes within a filesystem to log appends on the SSD
  - build drives out of arrays of SSDs, add lots of cache

SSD cost

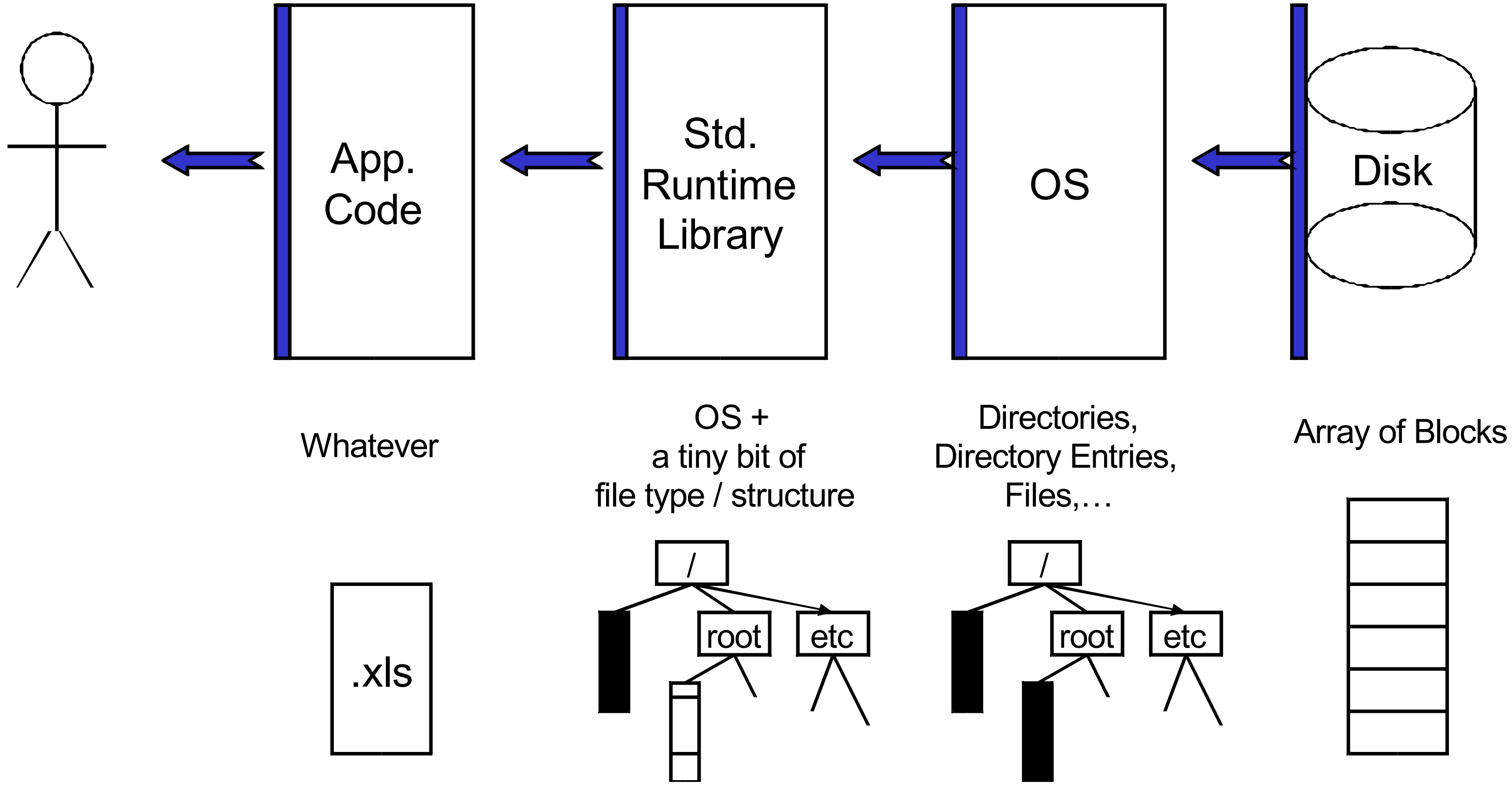
- **Capacity**
  - today, flash SSD costs ~$1.00/GB (down from $2.50 a year ago)
    - 1TB drive costs around $1000
      - 1TB hard drive costs around $100
  - Data on cost trends is a little sketchy and preliminary

- **Energy**
  - SSD is typically more energy efficient than a hard drive
    - 1-2 watts to power an SSD
    - ~10 watts to power a high performance hard drive
      - (can also buy a 1 watt lower-performance drive)

Interface Layers



Exported Abstractions



App. Code

Std. Runtime Library

OS

Disk

Whatever

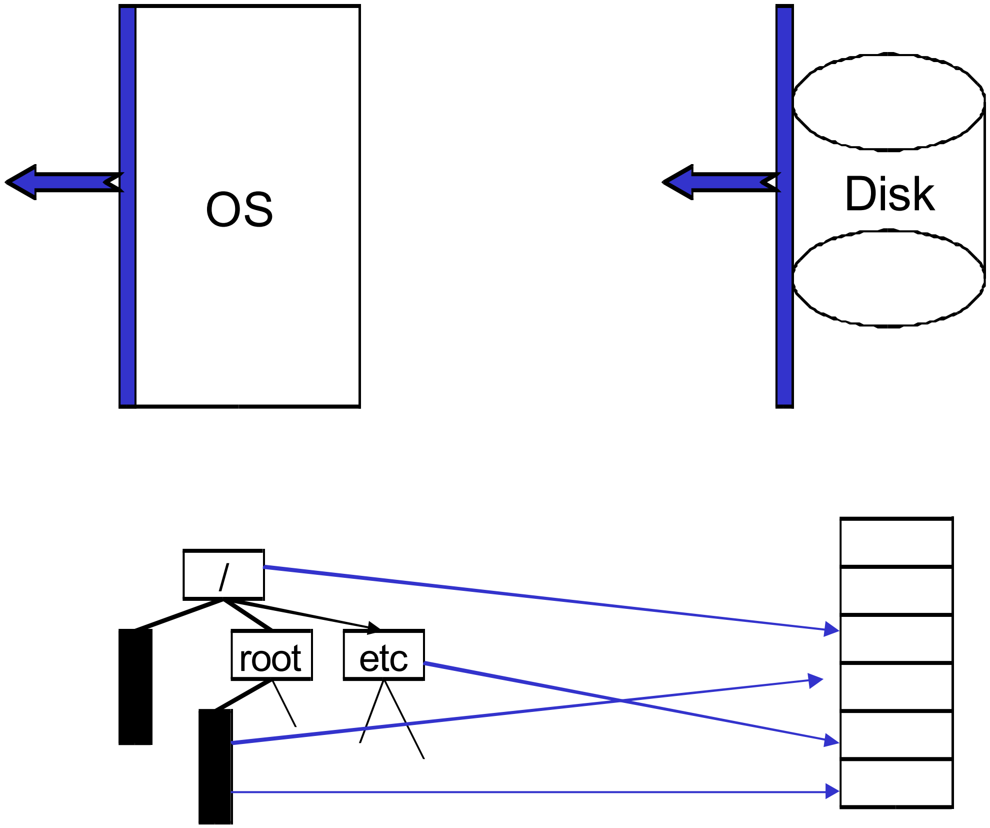
OS + a tiny bit of file type / structure

Directories, Directory Entries, Files,…

Array of Blocks

Primary Roles of the OS (file system)

1. Hide hardware specific interface
2. Allocate disk blocks
3. Check permissions
4. Understand directory file structure
5. Maintain *metadata*
6. Performance
7. Flexibility



File systems

- **The concept of a file system is simple**
  - the implementation of the abstraction for secondary storage
    - abstraction = files
  - logical organization of files into directories
    - the directory hierarchy
  - sharing of data between processes, people and machines
    - access control, consistency, …

Files

- **A file is a collection of data with some properties**
  - contents, size, owner, last read/write time, protection …
- **Files may also have types**
  - understood by file system
    - device, directory, symbolic link
  - understood by other parts of OS or by runtime libraries
    - executable, dll, source code, object code, text file, …
- **Type can be encoded in the file's name or contents**
  - windows encodes type in name
    - .com, .exe, .bat, .dll, .jpg, .mov, .mp3, …
  - old Mac OS stored the name of the creating program along with the file
  - unix has a smattering of both
    - in content via magic numbers or initial characters (e.g., #!)

Basic operations

Unix

- create(name)
- open(name, mode)
- read(fd, buf, len)
- write(fd, buf, len)
- sync(fd)
- seek(fd, pos)
- close(fd)
- unlink(name)
- rename(old, new)

Windows

- CreateFile(name, CREATE)
- CreateFile(name, OPEN)
- ReadFile(handle, …)
- WriteFile(handle, …)
- FlushFileBuffers(handle, …)
- SetFilePointer(handle, …)
- CloseHandle(handle, …)
- DeleteFile(name)
- CopyFile(name)
- MoveFile(name)

Directories

- **Directories provide:**
  - a way for users to organize their files
  - a convenient file name space for both users and FS's
- **Most file systems support multi-level directories**
  - naming hierarchies (/, /usr, /usr/local, /usr/local/bin, …)
- **Most file systems support the notion of current directory**
  - absolute names: fully-qualified starting from root of FS

```
bash$ cd /usr/local
```

  - relative names: specified with respect to current directory

```
bash$ cd /usr/local    (absolute)
bash$ cd bin           (relative, equivalent to cd /usr/local/bin)
```

Directory internals

- **A directory is typically just a file that happens to contain special metadata**
  - directory = list of (name of file, file attributes)
  - attributes include such things as:
    - size, protection, location on disk, creation time, access time, …
  - the directory list is usually unordered (effectively random)
    - when you type "ls", the "ls" command sorts the results for you

Path name translation

- **Let’s say you want to open “/one/two/three”**

  ```
  fd = open(“/one/two/three”, O_RDWR);
  ```

- **What goes on inside the file system?**
  - open directory “/” (well known, can always find)
  - search the directory for “one”, get location of “one”
  - open directory “one”, search for “two”, get location of “two”
  - open directory “two”, search for “three”, get loc. of “three”
  - open file “three”
  - (of course, permissions are checked at each step)
- **FS spends lots of time walking down directory paths**
  - this is why open is separate from read/write (session state)
  - OS will cache prefix lookups to enhance performance
    - /a/b, /a/bb, /a/bbb all share the “/a” prefix

File protection

- **FS must implement some kind of protection system**
  - to control who can access a file (user)
  - to control how they can access it (e.g., read, write, or exec)
- **More generally:**
  - generalize files to objects (the "what")
  - generalize users to principals (the "who", user or program)
  - generalize read/write to actions (the "how", or operations)
- **A protection system dictates whether a given action performed by a given principal on a given object should be allowed**
  - e.g., you can read or write your files, but others cannot
  - e.g., your can read `/etc/motd` but you cannot write to it

The original Unix file system

- Dennis Ritchie and Ken Thompson, Bell Labs, 1969
- "UNIX rose from the ashes of a multi-organizational effort in the early 1960s to develop a dependable timesharing operating system" – Multics
- Designed for a "workgroup" sharing a single system
- Did its job exceedingly well
  - Although it has been stretched in many directions and made ugly in the process
- A wonderful study in engineering tradeoffs

(Old) Unix disks are divided into five parts …

- **Boot block**
  - can boot the system by loading from this block
- **Superblock**
  - specifies boundaries of next 3 areas, and contains head of freelists of inodes and file blocks
- **i-node area**
  - contains descriptors (i-nodes) for each file on the disk; all i-nodes are the same size; head of freelist is in the superblock
- **File contents area**
  - fixed-size blocks; head of freelist is in the superblock
- **Swap area**
  - holds processes that have been swapped out of memory

So …

- **You can attach a disk to a dead system …**
- **Boot it up …**
- **Find, create, and modify files …**
  - because the superblock is at a fixed place, and it tells you where the i-node area and file contents area are
  - by convention, the second i-node is the root directory of the volume

i-node format

- User number
- Group number
- Protection bits
- Times (file last read, file last written, inode last written)
- File code: specifies if the i-node represents a directory, an ordinary user file, or a "special file" (typically an I/O device)
- Size: length of file in bytes
- Block list: locates contents of file (in the file contents area)
  - more on this soon!
- Link count: number of directories referencing this i-node

The flat (i-node) file system

- **Each file is known by a number, which is the number of the i-node**
  - seriously – 1, 2, 3, etc.!
  - why is it called “flat”?
- **Files are created empty, and grow when extended through writes**

The tree (directory, hierarchical) file system

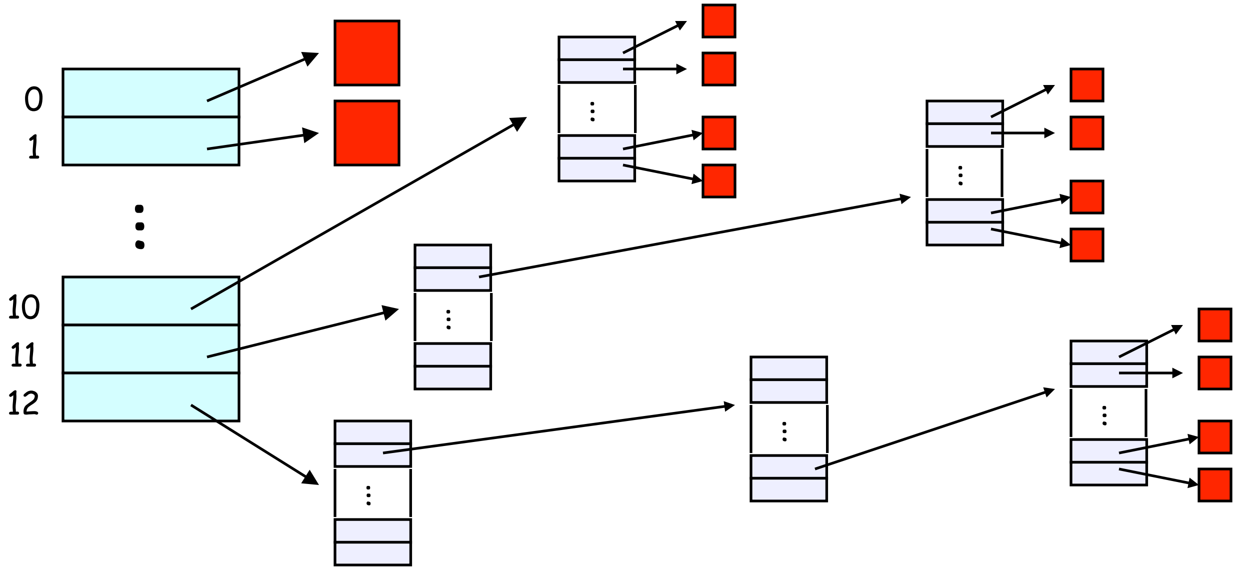
- A directory is a flat file of fixed-size entries
- Each entry consists of an i-node number and a file name

i-node number	File name
152	.
18	..
216	my_file
4	another_file
93	oh_my_god
144	a_directory

- It’s as simple as that!

The “block list” portion of the i-node (Unix Version 7)

- Points to blocks in the file contents area
- Must be able to represent very small and very large files. How?
- Each inode contains 13 block pointers
  - first 10 are “direct pointers” (pointers to 512B blocks of file data)
  - then, single, double, and triple indirect pointers



So …

- **Only occupies 13 x 4B in the i-node**
- **Can get to 10 x 512B = a 5120B file directly**
  - (10 direct pointers, blocks in the file contents area are 512B)
- **Can get to 128 x 512B = an additional 65KB with a single indirect reference**
  - (the 11th pointer in the i-node gets you to a 512B block in the file contents area that contains 128 4B pointers to blocks holding file data)
- **Can get to 128 x 128 x 512B = an additional 8MB with a double indirect reference**
  - (the 12th pointer in the i-node gets you to a 512B block in the file contents area that contains 128 4B pointers to 512B blocks in the file contents area that contain 128 4B pointers to 512B blocks holding file data)

And …

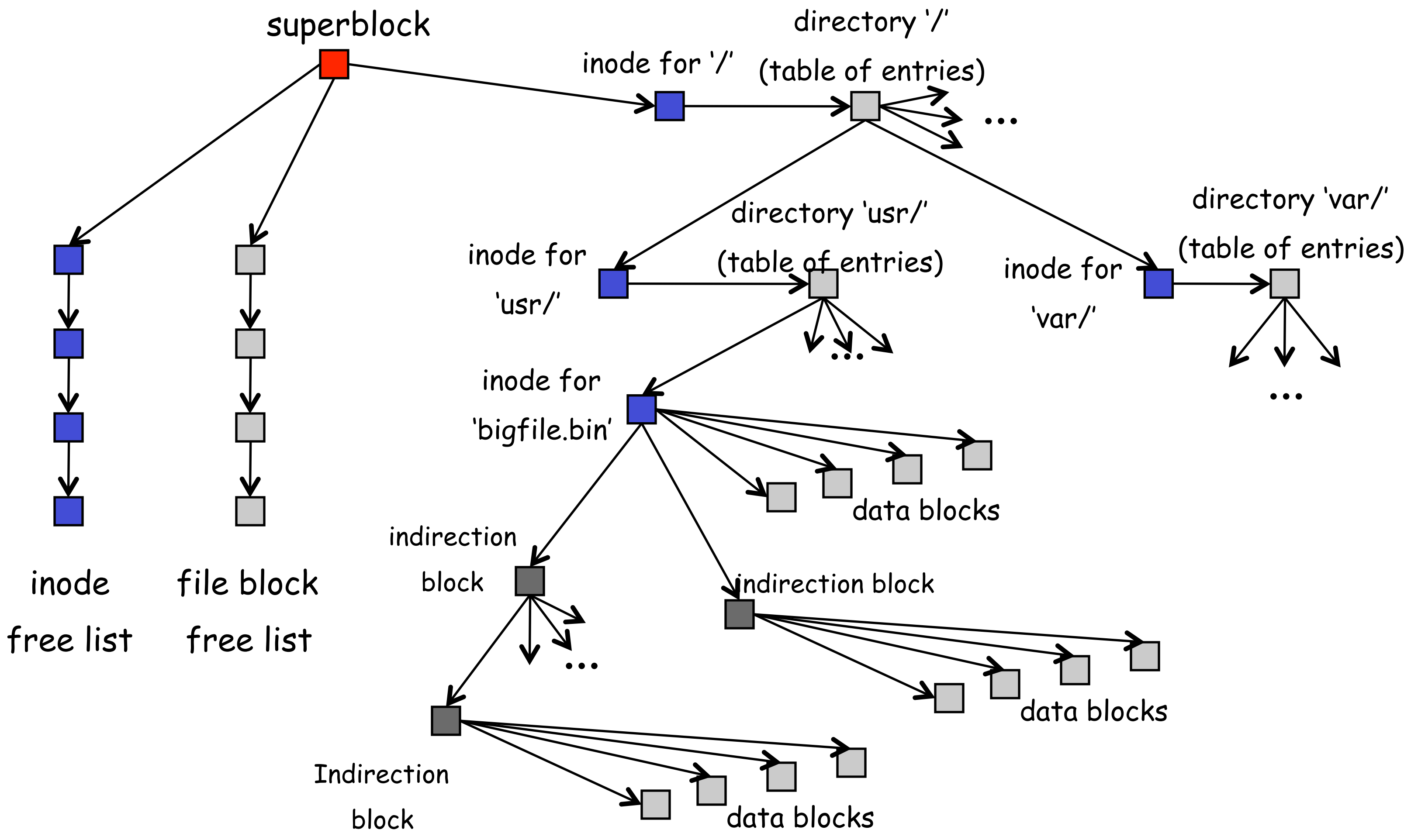
- **Can get to 128 x 128 x 128 x 512B = an additional 1GB with a triple indirect reference**
  - (the 13th pointer in the i-node gets you to a 512B block in the file contents area that contains 128 4B pointers to 512B blocks in the file contents area that contain 128 4B pointers to 512B blocks in the file contents area that contain 128 4B pointers to 512B blocks holding file data)
- **Maximum file size is 1GB + a smidge**

And then …

- **A later version of Bell Labs Unix utilized 12 direct pointers rather than 10**
  - Why?
- **Berkeley Unix went to 1KB block sizes**
  - What's the effect on the maximum file size?
    - 256x256x256x1K = 17 GB + a smidge
  - What's the price?
- **Subsequently went to 4KB blocks**
  - 1Kx1Kx1Kx4K = 4TB + a smidge

Putting it all together

- The file system is just a huge data structure



File system layout

- **One important goal of a filesystem is to lay this data structure out on disk**
  - have to keep in mind the physical characteristics of the disk itself (seeks are expensive)
  - and the characteristics of the workload (locality across files within a directory, sequential access to many files)
- **Old UNIX's layout is very inefficient**
  - constantly seeking back and forth between inode area and data block area as you traverse the filesystem, or even as you sequentially read files
- **Newer file systems are smarter**
- **Newer storage devices (SSDs) change the constraints, but not the basic data structure**

File system consistency

- **Both i-nodes and file blocks are cached in memory**
- **The “sync” command forces memory-resident disk information to be written to disk**
  - system does a sync every few seconds
- **A crash or power failure between sync’s can leave an inconsistent disk**
- **You could reduce the frequency of problems by reducing caching, but performance would suffer big-time**

What do you do after a crash?

- **Run a program called "fsck" to try to fix any consistency problems**
- **fsck has to scan the entire disk**
  - as disks are getting bigger, fsck is taking longer and longer
  - modern disks: fsck can take a full day!
- **Newer file systems try to help here**
  - are more clever about the order in which writes happen, and where writes are directed
    - e.g., Journaling file system: collect recent writes in a log called a journal. On crash, run through journal to replay against file system.

fsck i-check (consistency of the flat file system)

- **Is each block on exactly one list?**
  - create a bit vector with as many entries as there are blocks
  - follow the free list and each i-node block list
  - when a block is encountered, examine its bit
    - If the bit was 0, set it to 1
    - if the bit was already 1
      - if the block is both in a file and on the free list, remove it from the free list and cross your fingers
      - if the block is in two files, call support!
  - if there are any 0's left at the end, put those blocks on the free list

fsck d-check (consistency of the directory file system)

- **Do the directories form a tree?**
- **Does the link count of each file equal the number of directories linked to it?**
  - I will spare you the details
    - uses a zero-initialized vector of counters, one per i-node
    - walk the tree, then visit every i-node

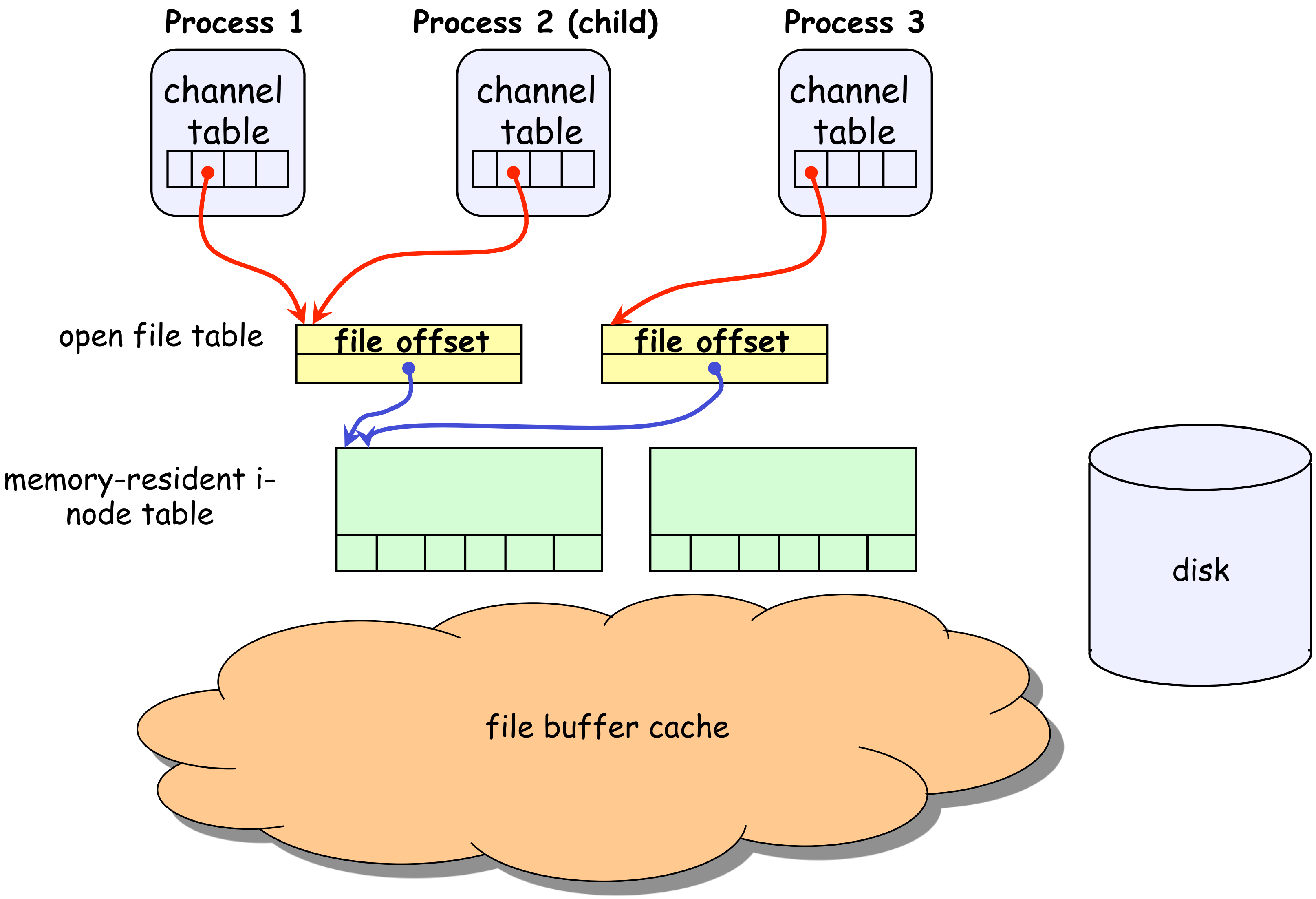
Protection

- **Objects:** individual files
- **Principals:** owner/group/world
- **Actions:** read/write/execute

- **This is pretty simple and rigid, but it has proven to be about what we can handle!**

File sharing

- Each user has a "channel table" (or "per-user open file table")
- Each entry in the channel table is a pointer to an entry in the system-wide "open file table"
- Each entry in the open file table contains a file offset (file pointer) and a pointer to an entry in the "memory-resident i-node table"
- If a process opens an already-open file, a new open file table entry is created (with a new file offset), pointing to the same entry in the memory-resident i-node table
- If a process forks, the child gets a copy of the channel table (and thus the same file offset)


Process 1
Process 2 (child)
Process 3
channel table
channel table
channel table
open file table
file offset
file offset
memory-resident i-node table
disk
file buffer cache